Notice of References Cited	Application/Control No. 10/780,466	Applicant(s)/Patent Under Reexamination MARVIN ET AL.	
	Examiner Mary J. Steelman	Art Unit 2191	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-5,604,860 A	02-1997	McLaughlin et al.	715/866
*	B	US-5,630,131	05-1997	Palevich et al.	717/108
*	C	US-6,018,730 A	01-2000	Nichols et al.	706/45
*	D	US-6,023,578 A	02-2000	Birsan et al.	717/105
*	E	US-6,470,364 B1	10-2002	Prinzing, Timothy N.	715/530
*	F	US-6,637,020 B1	10-2003	Hammond, Barton Wade	717/107
*	G	US-6,654,932 B1	11-2003	Bahrs et al.	715/507
*	H	US-6,789,054 B1	09-2004	Makhlouf, Mahmoud A.	703/6
*	I	US-7,017,146 B2	03-2006	Dellarocas et al.	717/106
*	J	US-7,062,718 B2	06-2006	Kodosky et al.	715/771
*	K	US-7,111,243 B1	09-2006	Ballard et al.	715/744
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Bugunovic, Nikola; "A Programming Model for Composing Data Flow Collaborative Applications", 1999 IEEE, retrieved 04/10/2007.
	V	Sung, SY; Soon, WM; Loh WL; Shaw, V" "A Multimedia Authoring Tool for the Internet", p. 304-308, 1997 IEEE retrieved 04/10/2007.
	W	Smith, Milton; Sodhi, Jag; "Marching Towards a Software Reuse Future", p. 62-72, Nov/Dec 1994, ACM, retrieved 04/10/2007.
	X	

*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

A Programming Model for Composing Data-Flow Collaborative Applications

Nikola Bogunovic
R.Boskovic Institute, Zagreb, 10000, Croatia
nikolabo@faust.irb.hr

Abstract

Distributed systems are essential for many real-word applications. The paper presents an experimental programming model that enables visual configuration, deployment and control of data-flow based collaborative systems, a class of distributed applications (DA). The programming model solves the problem of interoperability among DA functional components through the introduction of a fast middleware network software layer, and by implementing a transparent message-based communication between processes executing on machines connected by a local area network. At the next higher level, the programming model allows deployment and interconnection of encapsulated modules by logical composition of the entire collaborative application. The project differs widely from the existing systems in the network communication overhead, and the user interface design.

1. Introduction

The essence of heterogeneous computing is to maximize the performance by matching application's computational requirements with appropriate execution modes present in the underlying system. Any kind of programming model attempts to achieve the optimal matching of requirements to modes by the effective computational resource allocation. The assignment method depends on attributes of the underlying system.

Hierarchical distributed processing taxonomy [1] divides resource allocation schemes into static and dynamic, according to the time of decision-making. Static schemes are divided into optimal and suboptimal, depending on the availability of the set of input parameters and the cost function. Suboptimal schemes are further divided into approximate and heuristic. Dynamic schemes can be distributed or nondistributed depending on the authority of the allocation decisions. In the light of the described taxonomy the programming model of a class of distributed systems considered in this paper is *static, suboptimal and heuristic.*

Heterogeneous distributed systems can be supported at different levels: *network level/operating system level (OS)*, and *language level.* At the OS level a distributed application (DA) is implemented as a collection of sequential programs, which communicate using the relevant networking system calls. The communication interfaces are complex and difficult to use. The naming conventions and interprocess communication primitives are usually non-uniform, using different conventions and providing different semantics for internal and remote interactions. Little support is provided for the initial configuration of a set of program components, neither into an executable DA, nor for subsequent monitoring and control of the configuration. Applications programmed in this way are difficult to construct and maintain. The advantage of an operating system approach is very high flexibility. On the other hand, a distributed programming language approach reduces the complexity of constructing DA by providing modularity, concurrency, synchronization and communication facilities integrated into a single language framework. However, the integration results in a single large distributed program that is difficult to modify.

The programming model, implemented and analyzed in this paper combines and extends the simplicity and safety of a language approach with the flexibility of operating systems approach. On top of that, the model allows for visual configuration and reconfiguration of DA, separating the low level functional module design from the high-level DA composition task. The flexibility of our programming model leads to the loosely coupled distributed programs, focusing to the general peer (mesh) interconnections, rather than the exclusive client-server paradigm. Design goals follow a data-flow (DF) computation model [2], an abstraction denoted by a unidirectional graph, where nodes represent encapsulated functions and where arcs (links) represent the flow of data between functions. We have selected the pure data flow model, that is the model with no added control flow constructs. When all of a node's inputs are available, the node starts with data processing. A single, distinct node (the control node) assists in visual (re)configuration of the data-flow distributed model, and subsequently, during the run-time, supervises the entire application, Fig.1.

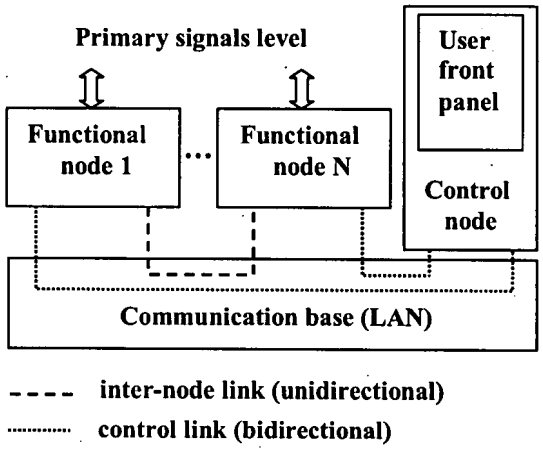

Figure 1. Generic data-flow collaborative model.

The succeeding section gives a brief survey of the current related work on distributed systems that support user composition at a certain level. The subsequent section presents in detail the proposed new network middleware layer that forms a hidden unifying software substructure for maintaining data-flow collaborative applications, with particular emphasis on performance issues. The description will follow bottom-up approach, starting with the functional node programming and proceeding on to the network software layer that integrates nodes into a DF computation model. The rest of the paper analyses an object-oriented architecture of the control node (CN) that supports DA organization and run-time supervision through the logical arrangements of graphical objects representing compositional modules.

2. Comparison to other related work

Several competing design principles necessary to build high performance and reliable distributed systems have evolved during 1980's and 1990's with tendency to represent middleware information infrastructure for heterogeneous distributed computing. They do not typically encompass any visual composition tools.

The *Open Software Foundation* (OSF) *Distributed Computing Environment* (DCE) [3], forms a middleware layer encompassing DCE *Executive* and *Extended Services*. DCE *Executive* consists of Security services, Directory services, Domain name service (DNS), Distributed Time service, Remote procedure call and Thread package service. DCE was designed to support procedural programming, but with additional capabilities that began to overlap with object oriented-systems. Clients request services through a well defined interface, specified by the *Interface Definition Language* (IDL).

Common Object Request Broker Architecture (CORBA) [4], is a competing technology that provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the user. Its design is based on object model that defines common semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. The CORBA runtime environment provides a richer set of object invocation and passing than the DCE environment. The central component of CORBA is the *Object Request Broker* (ORB) that encompasses communication infrastructure necessary to identify and locate objects.

Microsoft's *Distributed Component Object Model* (DCOM) [5], extends the COM model to support communication among objects on different computers. At the center of COM are mechanisms for establishing connections to components and creating new instances of components. The libraries provide sophisticated mechanisms for passing method parameters that build on the *Remote Procedure Call* (RPC) and *Interface Description Language* (IDL) infrastructure.

The *Java Virtual Machine* [6] allows a group of Java-enabled machines to be treated as homogeneous. Java makes it possible to dynamically load code in a Java running process. These features allow a system to invoke methods on remote objects from the calling process, which can also move code between the communicating distributed processes. Hence, *Java Remote Method Invocation* (RMI) supports distributed objects with new functionality and with a single language-centric design.

We have not selected any of the above technologies on the rationale that they do not yet support latency sensitive applications. Based on the analysis given in [7] the main sources of latency overhead in these systems arise from:

- long chains of intra-midleware function calls
- excessive presentation layer conversions
- non-optimized buffering algorithms
- inefficient server demultiplexing techniques
- lack of integration with OS and network features

3. Network based software architecture

The presented system clearly distinguishes between the programming of DA individual software components, and the integral system composition encompassing those components. In this section we give a description of DA nodes as the primary processing components, and continue with a report on the proposed network middleware structure that supports the node-to-node communication and user level DA composition.

The system recognizes two types of nodes: data-flow processing nodes (DFN) and a single control node (CN). Data processing nodes are self-contained sequential processes. These nodes exchange messages and perform functions such as signal processing, or a higher level data and information processing. The node procedure interface

is defined in terms of strongly typed unidirectional terminals that specify all the information required to use the module. The input and output terminals represent bindings to the UNIX Berkeley *sockets*, in accordance with the application programming interface to TCP/IP suite of protocols. The DA model presented in this paper assumes that all nodes reside on the UNIX class of operating systems, utilizing the necessary embedded standard system calls. All socket descriptors are given as file descriptors, simplifying data reading and writing over node's terminals. Two special bi-directional terminals of every node maintain communication with the control node (CN), utilizing descriptors 0,1,2, known also as *stdin*, *stdout*, and *stderr*. The node's communication links to other DFN nodes employ descriptors 3, 4, etc., Fig. 2.

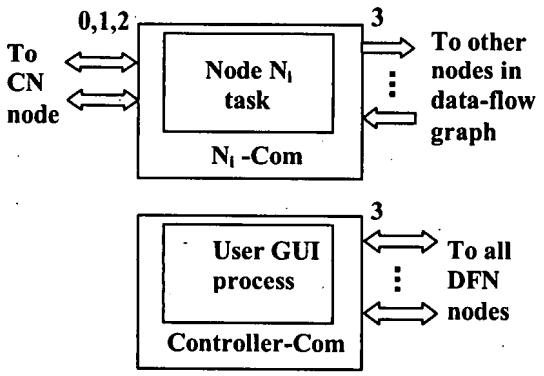

Figure 2. The structure of DFN and CN nodes.

At N_i node, regular data that are streaming from other nodes in the data-flow diagram, are captured, processed and forwarded by employing *read()* and *write()* functions. Control data, coming from the CN node, are accepted through the UNIX software interrupt structure. Fig. 3 gives the skeleton of a generic node function code.

After an initial processing, the main loop checks for the *signal* interrupt that may update some module parameters. The potential update is passed from the CN node over the terminal (descriptor) #0. Regular data are handled by simple routines that define the node functionality. Since terminals are unidirectional and strongly typed, the conversion from network representation and back must be accomplished.

To make the N_i node task programming as simple as possible, another associated process, N_i-Com, at each DFN node takes care of the communication complexity, Fig. 2. All N_i-Com processes are mutually identical and are installed to data-flow diagram nodes prior to the installation of node-specific functional procedures.

A data-flow diagram is composed at the CN node, and the information on each DFN node links is passed to the appropriate N_i-Com process to establish TCP connections. After creating links, the N_i-Com process starts the node task by using *exec()* system call, and exits. The *exec()* call passes all network links to the node N_i task program.

```
DFN module
begin
    associate(terminals)
    open_log(log)
    flush_log(log)
    initialize(data_buff)
    loop
        if signal
            new_params()
        get_data(terminal)
        compute(data)
        put_data(terminal)
    end
end

new_params()
begin
    read(0, &new_params, sizeof())
    convert(new_params)
    load(new_params)
    set(signal)
end

get_data(terminal)
begin
    loop
        read(terminal, &data, sizeof())
        convert(data)
    end
end

put_data(terminal)
begin
    format(data)
    write(terminal, &buf, sizeof())
end
```

Figure 3. Generic node N_i task program.

Any data-flow diagram is uniquely characterized by a list containing for each DFN node:

- the host name
- the path to the N_i-Com program
- the path to the node N_i task program
- the number and id of links to other nodes
- definitions of sending and receiving terminals

The control node (CN) encompasses two processes, Fig. 2: the multifaceted user GUI process that will in the pre-processing stage create the above data structure specifying the data-flow diagram, and the Controller-Com process that will interpret this data structure and instruct each N_i-Com process to establish TCP links. Upon setting up the working data-flow system; the Controller-Com will turn the command back to the user GUI process.

The communication between the user GUI process and Controller-Com is achieved through UNIX pipes. The communication between Controller-Com and each N_i-Com is implemented over predefined descriptors (0, 1, 2) as the exchange of lines that contain printable characters.

After each N_i-Com acknowledges that it has been able to open all *socket* handles, Controller-Com issues two different commands to two N_i-Com processes at the boundaries of the link:

```
accept <line #>
```

that returns a port number which is then passed to the sending N_i-Com with the message:

```
connect <line #> <host> <port #>
```

The accept-connect procedure is repeated for each TCP link in the data-flow diagram until all links are established. Next, the Contoller-Com process instructs each N_i-Com to activate the corresponding node task by the *exec()* system call. N_i-Com processes will then exit.

Subsequently, The Controller-Com process will: 1) start user GUI process as the DA visual monitor, 2) open pipes to it, 3) pass all DFN node descriptors (terminals), 4) put itself in the "stand-by" state waiting for the end message that may come through the pipe from the user GUI process. It is absolutely necessary to pass the data-flow connections (DFN links) through the pipe, since the user GUI process cannot inherit them (they are not in the scope of its parent process).

The described communication complexity is completely hidden from the user, who must just provide the node functionality modules (executable files designed according to the template given in Fig. 3), and the data-flow composition structure as an unambiguous list of nodes and links.

4. Performance issues

The implementation of a system of such a complexity and granularity subsumes constraints that originate in the operating system (OS) and networking. OS context switching overhead significantly impacts the performance and predictability of any one multithreaded process, and indicates the efficiency of the kernel dispatcher. The measurement is not easy and must encompass voluntary (thread waits for a resource) and involuntary (thread is blocked due to a higher priority thread) context switch. Some recent results that affect ORB requests are given in [8] and partly reproduced in Table 1.

The S-R test is the suspend-resume test that measures an average switch between two blocked threads to resume operation. Linux OS does not support S-R interface. The yield test shows the switching of two threads to call a system function. The synch S-R test measures switching between high and low priority threads on a mutex (semaphore) synchronization. The results demonstrate that the voluntary switching times in the range of 10μs do not contribute significantly to the overall latency.

Table 1. OS context switching times [μs].

OS	S-R	Yield	Synch
Win NT	1.41	1.77	2.66
Solaris	21.30	11.20	131.20
Linux		2.60	9.72

Good overall system performance requires that the statistically dominant, higher level operations map efficiently into kernel primitives. The communication speed between the DA nodes depends on their location. If the DA nodes reside on the same host, the TCP/IP overhead may be assumed constant. If the DA nodes reside on hosts connected through the Ethernet (CSMA/CD protocol), there is a stochastic parameter in the communication path. Approximate methods for modeling CSMA/CD algorithms [9] assume that the packets carrying the original traffic and the retransmission packets are forming a Poisson stream. In the model the throughput is given by the factor S, and the system is solved as $S = f(load)$. Since the load is again the function of throughput and retransmissions r, we obtain:

$$S = f(r\ S)$$

Given a geometrical distribution of retransmissions, and allowing the maximum number of 16 attempts, one can find the packet dropping probability that has the profound effect on the expected delay. The dropping probability is small (indicating a constant network delay), but rises sharply above 0.7 of the maximal network load.

Recent empirical investigations of the network traffic show that it has a fractal (self-similar) nature rather than Poissonian. The consequence of this discovery is that most analytical models do not apply to the real world situation. Hence we will produce some quantitative data for the considered data-flow architecture implemented over a 10Mb Ethernet LAN, and utilized below 0.7 of the maximal load.

Table 2. Data transfer times [ms].

Bytes	Local	Remote
64	0.05	0.50
256	0.20	1.50
1024	0.75	4.50

Table 2 gives the time for a DFN node to copy different amounts of data both locally and remotely. The table reflects transmission times to copy the bytes between the memory locations. It indicates the best performance the hardware can support. The most of remote transmission

times are lost on the latency, rather than on the actual network transfer.

Table 3. Data read file transfer times [ms].

Unit	Local	Remote
64KB	60.00	280.00

Table 3 takes into account file access times. Once the sockets are opened, the entire data transfer is achieved by `write()` and `read()` system calls. The performance measurements indicate the dominant role of disk access times. Without disk read-ahead the cost of a remote file read is basically the sum of the block request time and the disk access time. The absence of network read-ahead introduces another performance penalty, but only if the client is reading data blocks slower than the server is delivering them. We conclude that the network read-ahead is of no performance benefit when the file access time is dominated by the disk access time.

5. Data-flow diagram composition support

It was already noted that the data-flow diagram is uniquely specified with a list encompassing all nodes (hosts with paths to their N_i-Com and task programs) and links (with terminals at both ends). However, manual specification of the list can be a very laborious job, without automated, built-in logical error correcting facilities. We have devised a novel graphical environment that supports intuitive data-flow diagram composition and generation of the consistent configuration list.

Many software packages have been developed that support device-independent interactive graphics on top of the UNIX X11 Window system core libraries (Xlib and Xt). Structured graphics packages manipulate elements of a display list (may be lists themselves) making it possible to compose hierarchies of graphical elements, but the libraries are large and monolithic, and difficult to extend. We have selected InterViews system [10], an object-oriented C++ library that resolve the problems of structured graphics by data hiding and protection, extensibility and code sharing through inheritance, and flexibility through runtime binding of operations to objects. Composition mechanisms are central to the design of InterViews. Primitive and composition objects are linked into the application code. The application's user interface is defined in terms of InterViews objects, which communicate with the window and OS. Three categories of objects (interactor, graphic and text) that supports composition are implemented as hierarchy of object classes derived from a common base class. Building compositions that combine simple behavior can specify complex behavior. The composition protocol facilitates the task of both the designer of a user interface toolkit and the implementor of a particular GUI. To further expedite the design of a user interface that employs many (hundreds of) objects the latest InterViews implementation encompasses, as an extension, simple and efficient objects called "glyphs". The `Glyph` base class defines a protocol for drawing: subclasses define specific appearances such as graphic primitives (characters and spaces), and compose objects. Applications define their appearance by building hierarchies of glyphs.

We were interested in designing a graphical editor that allows the user to manipulate graphical representations of familiar objects directly. The editor must provide animated feedback as the user creates and manipulates graphical objects. Unfortunately, editors are difficult to build with general interface tools because of their special requirements. The InterViews library concept offers Unidraw, a collection of programming abstractions that simplifies the construction of graphical object editors by defining four basic abstractions: (1) *components*, (2) *tools*, (3) *commands*, and (4) *external representations*. Unidraw also supports multiple views, graphical connectivity, and dataflow between components. Sadly, Unidraw was based on an obsolete InterViews version that did not support the `Glyph` class. Since a new project, InterViews based and CORBA compliant, Fresco object environment [11] has not yet been completed, the TinIV programming abstractions [12] appeared to be an adequate substitution.

The final data-flow composition environment is presented in Fig. 4a. From the user perspective, the system embodies three distinctive segments: the toolbox, Fig. 4a, left, the data-flow diagram editor (DFDE), Fig. 4a, center, and the front panel layout editor (FPLE), Fig. 4a, right (to be explained later). Within DFDE, each class of graphical objects, that represents a DFN node, has a specific shape to signify the node's functionality.

A generic module has a rectangle shape with attached unidirectional terminal points. Hierarchical module structuring, as a method of complexity and restricted display area management is achieved by a class of *composite* modules, whose functionality is defined by their embedded parts. Additional terminal points for a composite module are still needed, though. Various tools (*select, move, line-connect*, etc.) achieve direct manipulation of all graphical objects, through *object handles*. The tool *examine* enables interactive set-up and change of all non-visible object attributes (e.g. default values of graphical objects), Fig. 4b. In addition to direct graphical object manipulation tools, the system embodies a set of *commands* that enable operations like: *new, cut, generate, run*, etc.

DFDE enables a system designer to select front panel graphical objects (*scales, displays*, etc.) and connect them to data-flow nodes via terminal points. Front panel objects

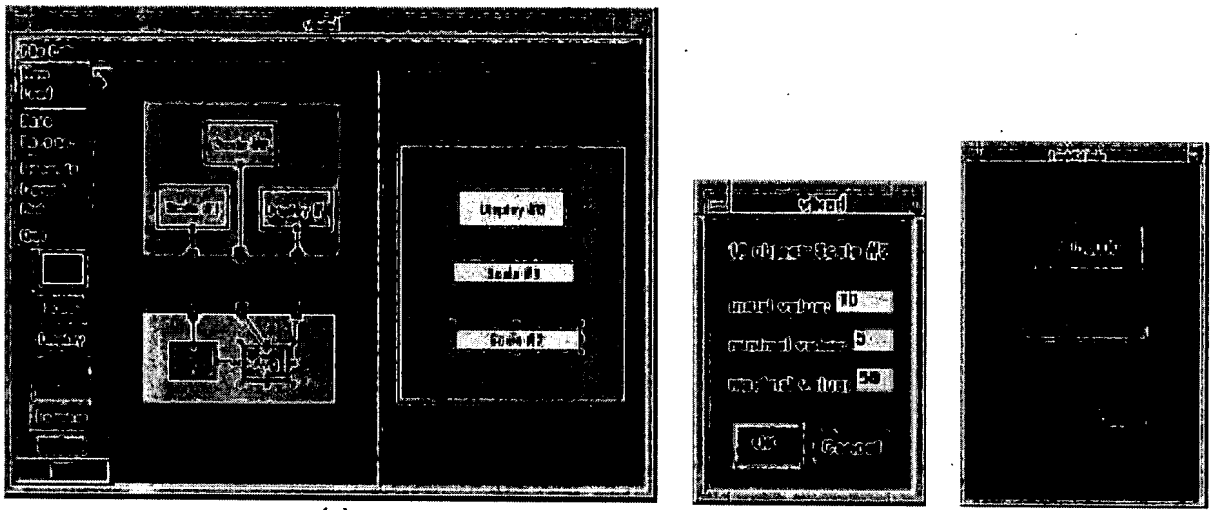

Fig. 4. Graphical environment for composing a data-flow distributed application.

are generally not connected between themselves, but rather through terminal points to the processing modules.

The presented environment relies heavily on the TinIV classes, both original and newly derived. The main problem was in the definition of multiple embedding. Originally, all objects had to be contained within the objects of the class `GraphicMaster`. Modification of primary methods `Graphic::contains` and `Graphic::intersects`, together with the revision of some auxiliary methods and files, corrected the problem. Direct object manipulation, within the entire editor hierarchy, is achieved by a recursive search to the bottom-intersecting rectangle that encompasses the defined point. Some new classes had to be derived from the three TinIV main classes: `Graphic`, `Tool`, and `Cmd`. In accordance with the mechanism of run-time class identification within the TinIV library, each newly derived class includes a unique symbolic and numeric indicator, and redefines methods `Graphic::get_classid()`, and `Graphic::is_a()`.

6. Controlling the data-flow application

The example, presented in Fig. 4, depicts a simple DA that embodies in its data-flow diagram two nodes (within the *composite* container): a process data acquisition node *Gen*, and a data averaging node *Avg*. All corresponding terminal points are of the same data type. The control panel (user interface) contains two *Scale* graphical objects (for setting up acquisition and averaging rates), with minimal and maximal values as their hidden attributes, and a single *Display* object, that shows the result of data processing to a desired precision (object's hidden attribute).

As soon as the system designer places a front panel graphical object in the upper part of the DFDE area, the same object is copied to the FPLE work area, Fig 4a, right. If the object is deleted from the DFDE, it is also deleted from the FPLE area. The user (in the FPLE area) drag objects, and composes the particular and unique front panel look-and-feel. When the design of a data-flow distributed application, together with the graphical interface to it, is finished, the designer invokes a compile process that will generate: 1) a complete data-flow configuration list, and 2) a virtual front panel to the application, Fig. 4c. Finally, the *run* command, given from the composition environment, will invoke the data-flow configuration process, as described in section 3, and transfer the control to the newly designed front panel.

During the generation process, the `cc` compiler utilizes the OSF/Motif library of low level graphical objects. Hence, the integral data-flow composition and run-time system employs a significant number of libraries interrelated through layers as depicted in Fig. 5.

7. Conclusion

The paper reveals that the composition and control of data-flow distributed application through an intuitive graphical environment is a complex process that integrates many and diverse computational paradigms. The presented system clearly separates the programming of DA individual software components (modules, nodes), and the system composition encompassing those components. Evidently, much more effort was placed on

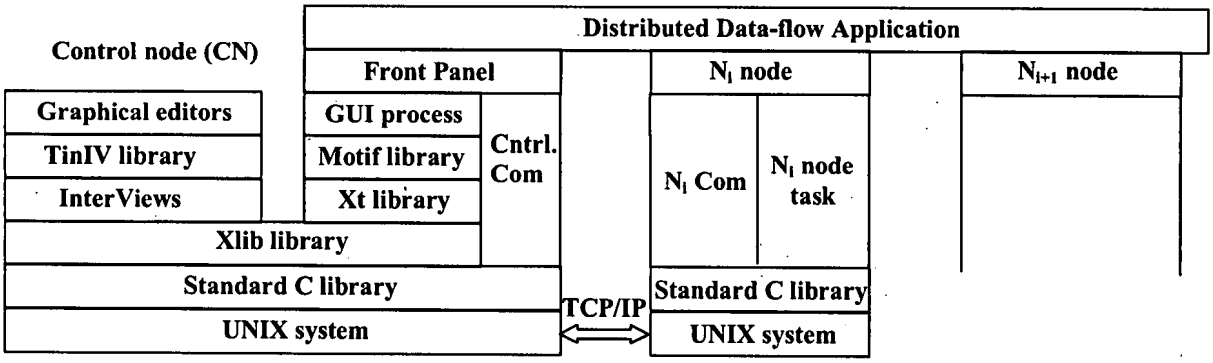

Figure 5. Layered libraries in composing and running a data-flow application.

the programming in the large (the composition task) then on the *programming in the small* (module design), which should be accomplished using low level programming paradigms.

The data-flow diagram is mapped to an underpinning software layer that provides node-to-node communication paths. The design objectives of an easy (re)configuration and fast data transmission are conflicting, but we believe that our system represent a balanced approach even for the near real-time applications. The present Ethernet type of LAN invalidates the design approach steering towards hard real-time systems. The lack of documented performance measures for the competing systems makes it difficult to place the described approach in the right comparative perspective.

User interface software is inherently difficult to create, because it combines some of the most difficult aspects like multi-processing, real-time programming, the need for robustness, and frequent, iterative changes to the specification. As a consequence, the area of user interface tools is expanding rapidly. Tools coming out of research labs are covering increasingly more of the user interface task. The presented visual multipurpose composition environment reflects the need for a well-documented and standardized library of graphic objects. This conclusion is readily inferred by inspecting Fig. 5.

8. References

[1] I.Ekmecic *et al.*, "A Survey of Heterogeneous Computing: Concepts and Systems", *Proc. of the IEEE*, vol. 84, No. 8, August 1996, pp. 1127-1143.

[2] J.Herath et al., Data-flow computing models, Languages, and Machines for Intelligence Computations, *IEEE Trans. on Software Eng.*, Vol.14 (1988), No.12, pp.1805-1828.

[3] W.D.Rosenberry, D.Keney, and G.Fisher, *Understanding DCE*, O'Reilly& Associates, 1992.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, February 1998.

[5] Microsoft Corp., *DCOM Architecture*, White Paper, One Microsoft Way, Redmond, WA, 1998.

[6] T.Lindholm and F.Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Longman, Reading, 1996.

[7] A.S.Gokhale and D.C. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks, *IEEE Transactions on Computers*, Vol. 47, No. 4, April 1998, pp.391-413.

[8] D.L. Levine *et al.*, "Measuring OS Support for Real-time CORBA ORBs", Submitted to the 4th IEEE Workshop on Object-oriented Real-time Dependable Systems, Santa Barbara, CA, Jan. 27-29, 1999.

[9] K.Sohraby, M.L. Molle, Comments on Throughput analysis for Persistent CSMA Systems, *IEEE Trans. On Communications*, Vol. COM-35, No.2, pp 241-243, February, 1987.

[10] M.A.Linton et al., *InterViews Reference Manual Version 3.1*, Stanford University, 1992.

[11] X Consortium, Inc., *Fresco Specification*, April 1994, Working Draft, ver 0.7.

[12] C.Wagner, TinIV - this is not InterViews, structuierte graphic für InterViews 3.1, Technical Report, Univeristät/GHS Siegen, FB 12 - Technische Informatik, January 1994.